# CG2111A Midterm

AY24/25 sem 2

github.com/mendax1234

## GPIO Programming

1. **Pin Mapping Table**

| Arduino Pin | Atmega 328 port and pin number |
|---|---|
| 0 | Port D, pin 0 |
| 1 | Port D, pin 1 |
| 2 | Port D, pin 2 |
| 3 | Port D, pin 3 (OC2B) |
| 4 | Port D, pin 4 |
| 5 | Port D, pin 5 (OC0B) |
| 6 | Port D, pin 6 (OC0A) |
| 7 | Port D, pin 7 |
| 8 | Port B, pin 0 |
| 9 | Port B, pin 1 (OC1A) |
| 10 | Port B, pin 2 (OC1B) |
| 11 | Port B, pin 3 (OC2A) |
| 12 | Port B, pin 4 |
| 13 | Port B, pin 5 |

2. **Source and Sink Current**
   - **source current**: the current flows **out of** the pin
   - **sink current**: the current flow **into** the pin

   For the specific characteristics, see data sheet P365-P366.
3. As long as the pin you want is set to 1, it will output HIGH, e.g. the LED will be turned on if it is connected.
4. **Output or Input**: **Output or Input** regards to the Arduino itself. **Know whether it is output or input before doing anything!** 0 is used to configure input and 1 is used to configure output.
   - **Read the Input value**: To read the value from an **input** pin, `PINB, PINC or PIND` contains the value you want.
5. **Bit Position**: The leftmost bit is Bit 7, the rightmost bit is Bit 0.
6. Be aware of the **active-low** / **active-high** characteristic of the electrical component, and then think about what value you want to set to your output Arduino Pin.
7. **Hex to Binary**

| Hex | Binary |
|---|---|
| A | 1010 |
| B | 1011 |
| C | 1100 |
| D | 1101 |
| E | 1110 |
| F | 1111 |

## Interrupts

1. **External Interrupts for Digital I/O**
   - **INT0** and **INT1** are external interrupt pins (**PD2** and **PD3** on the ATmega328p).
   - They are typically used for detecting changes in **external signals**, such as **button presses or sensor readings**.
   - **INT0** and **INT1** can be triggered on specific **edge transitions** (rising or falling) or **low level signals**.
2. **Interrupt Mask: `cli()` and `sei()`**
   - **`cli()`: Clear Interrupts**
   - **`sei()`: Set Interrupts**

   These two functions are used globally in for example `setup()` in the ATmega328p.
3. **`volatile` keyword**
   - TLDR, Always declare GLOBAL variables that are changed by ISRs to be `volatile`
4. **Calculating Transfer Time**
   - **Polling**: The key for polling calculation is to calculate the **total clock cycles** the CPU spends in the **polling loop** while waiting for each byte to be ready. (a.k.a, **In Polling, we are more interested in the cycles wasted between reading each byte**) We ignore the cycles taken to read each byte.
   - **Interrupts**: The key for interrupt calculation is to calculate the **total clock cycles** required to **handle each interrupt**, which typically **processes one byte**.
   - **DMA**: The key for DMA calculation is to calculate the CPU cycles spent on setting up the DMA transfer, which can then transfer multiple bytes without further CPU intervention.
5. **Every pin has an Interrupt, but not each pin has an unique interrupt**. The 23 `PCINT` are grouped into 3 groups and all lines in the same group will trigger the execution of the **same** ISR.
6. **Nested Interrupt**:
   - It is **not enabled by default**
   - If **enabled and triggered**, the interrupt may hand control back to ISR (not necessarily to `main`).
7. **Interrupt Priority**: The **lower the address the higher the priority level**.
8. **Context Switch**: Always remember to consider the **context switch**!

## PWM

1. When the question has **PWM**, the Timer/Counter Module is by default using **Phase-Correct Mode**.
2. The **default unit** for **period** is **seconds**. This means when you convert period to frequency, you must convert period to seconds first.
3. In Phase Correct Mode, the `OCRnA` value is **always updated** when the TC counts to **TOP**.
4. The two output compare pins can be set to generate PWM waveform together. e.g., set `OCR0A` will set the duty cycle for `OC0A`, set the `OCR0B` will set the duty cycle for `OC0B`.
5. **Always check the `TCCRnA/B`'s `COM0A/B[1:0]` mode and draw the corresponding waveform to derive the duty cycle! Since it may affect the duty cycle!!!**

## Timer

1. When the question has **Timer**, the Timer/Counter Module is by default using **CTC Mode**.
2. **Timer Resolution vs. Timer Frequency**
   - The **timer frequency** ($\frac{F_{clk}}{P}$) tells you **how many times the counter increments per second**. Its unit is **Hertz** (Hz).
   - The **timer resolution** ($\frac{1}{\text{Timer Frequency}}$) tells you **how much time each increment takes**. Its unit is **seconds** (s).
3. **Interval**: Interval refers to the **time period between consecutive events** or **triggers**.
4. **Waveform Period/Frequency calculation**
   - When you know `OCRnA` value and want to use it to calculate **period/frequency of the waveform** you want to generate, **+1 and then times the timer resolution**.
   - When you know the **period/frequency of the waveform** you want to generate and want to use it to calculate the `OCRnA` value, **-1 and then times the timer resolution**.
5. **The intermediate value** $V$: $V - 1 =$ `OCRnA`. When rounding the $V$ value, we always **round up**.

## ADC

1. **Nyquist Theorem**: Signals should be sampled with **at least twice as fast as the highest frequency content of the signal**.
2. **ADC Sampling Rate**: The **ADC sampling rate** refers to the **number of samples per second**. It is also called the **sample rate**. The unit of sampling rate is **samples per second** or **Hertz**. However, we know that **Hertz is the unit of frequency**. Therefore, the **sampling rate** can be also referred to as **sampling frequency**.
3. In the **PRR**, the **PRADC** should be written to 0 to enable the ADC conversion!

## USART

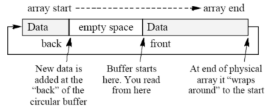
1. **Parity Bit**: The UART parity bit is set up when the **transmitting device is preparing to send data**. It is **not set in the receiver**. The receiver just receives the data the transmitter sends and **uses** the parity bit to check whether an error occurs.
   - **2D Parity Bit**:

| Data bits -> | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 | Parity (within byte) |
|---|---|---|---|---|---|---|---|---|---|
| Byte 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | **1** |
| Byte 2 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | **0** |
| Byte 3 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | **0** |
| Parity (across bytes) | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** 0 |

   - The last bit of the **parity (across bytes)** is calculated **across the last column, not using the last row!**
   - When looking at such table, find **which column and which row** may be wrong, if there are multiple rows and columns that are wrong, you can **only change one bit in a row or column once**!
2. **Sending Data Frame using UART**: **what you send is what you get**. e.g. You send the start bit first, then the first bit received by the receiver will be the start bit.
3. **UART Setup preparation**:
   - Number of data bits - 5, 6, 7, 8 or 9. Standard is 8.
   - Type of parity bits: None(N), Even(E), or Odd(O)
   - Number of stop bits: 1 or 2.
   - Bit rate: 1200, 2400, 4800, 9600, etc.
4. **Baud Rate and Bit Rate**:
   - **Baud rate**: **The number of symbols** that can be sent per second.
   - **Bit rate**: **The number of bits** that can be sent per second.
   - **Conversion**: Bit rate $=$ Baud rate $\times$ bits required to represent per symbol
5. **Circular Buffer**: It is the method we use to implement our buffer in USART.

Circular Buffers

- **The diagram below shows a circular buffer:**
  - Data is inserted using the "back" pointer.
  - Data is read using the "front" pointer.
  - Pointers are incremented using modulo:
    - ✓ **back = (back + 1) % SIZE**
    - ✓ **front = (front + 1) % SIZE.**
  - We will make use of a counter to decide when the buffer is empty or full.




## Misc

1. **Unit Change (seconds)**:

| Unit | Seconds Equivalent |
|---|---|
| µs (microsecond) | $10^{-6}$ s |
| ms (millisecond) | $10^{-3}$ s |

2. Arduino **doesn't have an OS**.
3. If you left shift a 1, for example by 10 position, and store the result into an 8-bit register. The final value will only **take the last 8 bits.**
4. For a normal 8-bit register, if you increment the value in it by 1 all the time, it may wrap around or may not.